Connectionist lexical processing

Stoianov, I

Chapter 3

CONNECTIONIST MODELING

3.1 Foreword

It is not strange that evolution has developed a completely different sort of a computational device than the common computer, which has a single processor (CPU) and shared memory. Everyone working with computers has experienced unpleasantness when his/her computer crashes and loses data. Moreover, this might happen even to the most reliable of such single-processor systems. Fortunately, the nervous systems of all animals on our planet, including ourselves, do not crash not only in normal conditions, but also in extreme situations, such as moderate loss of neurons. The main reason for this reliability is that the neurons – the main building blocks of the nervous systems – work independently of each other and in parallel. Even if some of them fail, the rest still correctly process the incoming information, thus most probably controlling the organism correctly. This parallelism also derives huge computational power out of slowly working chemical processors. Unfortunately, however, most of the current computers rely on one processing unit only, whose crash would be fatal for the life of a working system. The computational power of those linear computers is also limited by physical properties of their processing units. For example, the maximal CPU processing speed – the most critical productivity factor – is almost attained in current CPUs running at about 1 GHz.

In this chapter I will briefly introduce the basics of an alternative to the classical computational architectures, which is inspired by the neural systems of the living organisms. This alternative is *connectionism*, also

known as artificial neural network systems, which has been popular for more than two decades. In the remaining of this thesis I will present my ideas about how connectionism can be used to model one aspect of the human cognitive capacity – the phenomena of natural language.

The chapter starts with a short comparison of two main computational paradigms used nowadays – linear vs. parallel, connectionism being a special subclass of the latter. Then, background information about biological neurons and neuronal structures will be presented, which provide the main inspiration for the developments in connectionism. Basic information about artificial neurons and neuronal structures is presented in the next section, which is followed by a discussion about the way the information may be represented in connectionist models. I also present a few established connectionist models, focusing on dynamic neural networks, which are most adequate for linguistic processing. Special attention to the so-called Simple Recurrent Networks (SRNs) will be paid in the last section 3.7, which is used as a basic neural network throughout the remaining chapters.

Those readers that are already conversant with connectionism might skip this introductory chapter and only use section 3.7 as a reference about SRNs. It will be rather necessary for the audience with linguistic background who may still feel uncomfortable with the connectionist ideas and terminology.

3.2 Computational paradigms

In this section I introduce two basic computational paradigms used today – linear and parallel computing, stressing some advantages of the latter. Then, the connectionist paradigm will be introduced and will be shown to offer a very reliable method for computations and modeling.

Linear vs. Parallel computations

There are two very different ideas about how computing should be done: linear and parallel. *Linear computing* is what most computers today do: a single central processing unit (CPU) performs one operation at a time. Those operations are successively taken from a written algorithm (program) stored in a special common memory. This model corresponds to the classical von Neumann (1903 - 1957) architecture. Although it offers a simple method for computation, linear computing is vulnerable to crashes due to the single processing unit which as a physical device is subject to failure. Another disadvantage of a single processing machine is the processing speed, which is directly dependent on the speed of the processor as I already mentioned.

On the other hand, *parallel computing* is based on systems consisting of more than one processing unit, which together perform more than one operation at a time. Those processing units have their own local memory and may work independently of each other. A parallelism at such a physical level directly addresses the problems of limited speed and volnurability to system failure. Still, in order to perform a global task, synchronisation mechanisms are necessary, such as common memory and special signalling. In spite of the parallel processing, a centralisation of the control again makes the systems vulnerable to crashes and is the bottleneck of the speed of parallel systems. However, there are other types of parallel systems – connectionist systems – that exploit the principles of parallel processing even further – at a data level – which increases the reliability even more.

Connectionism / Parallel Distributed Processing

Connectionism refers to a special sort of parallel processing, where densely interconnected processing units perform very simple operations – signal accumulation and transmission – and the memory is represented as activation of the units and the strength of the connections between them. Those connections are modified according to very simple learning mechanisms, which are usually local in time and space.

External signals enter the system at some pre-specified units which have interface functions only and which distribute their activation to other processing units. In turn, when those processing units accumulate critical amount of signal, they produce impulses which are propagated to yet other units, etc. The activation of some of the units in the system is interpreted as a product of this system (see Fig. 3.5) and it might directly activate external devices (effectors). In living systems effectors are muscles and glands; in the artificial systems effectors are motors, computer displays, and so on.

The capacity to adjust the reaction of the connectionist systems adaptively in response to the current input, according to its "task" is what makes them very appropriate for modeling. They can *learn* almost any desired function if a pre-specified behaviour is required, or an input signal needs to be converted so that a consequent processing would process more easily the processed signal.

Further, in such distributed systems there is no central single controlling processor. Some neuronal sub-systems might be involved in some sort of higher-level controlling mechanisms, but even then control is distributed. No matter what the function of a neuron in such a system is, each neuron performs the very simple tasks of signal gaining, reacting and synapse ad-

justing, which makes the connectionist systems very reliable and resistant to failures of single neurons.

This kind of processing is also called *Parallel Distributed Processing* (Rumelhart, Hinton & Williams 1986). This name emphasises one very important feature of connectionism, namely that the signal being processed is distributed among a number of units. Beside increasing the reliability of the system, this peculiarity induces inherent semantics of the distributively represented data being processed. In contrast, the classical symbolic systems work with symbols which do not have inherent meaning. The meaning there is externally attached. In a PDP system, such as connectionist image processing, data is represented with its content and each of its elements (here pixels) is processed in parallel by one neuron. Similarly, linguistic data might be represented with vectors in which each element stands for a certain feature. It is also possible that the semantics of those features overlap, which redundancy further increases the reliability of the processing. Further, input data which shares most of their features would naturally be processed in a similar way, which leads to another very important property of connectionism, namely *generalisation* – the capacity correctly to process unseen data. If the model has learned how to process a certain class of data by examples, other unseen examples from this class will most probably be processed in a similar way due to the shared features.

3.3 Biological Neuronal system. Organisation.

The idea of Parallel Distributed Processing (PDP) is inspired by the organisation of the brain, which consists of about 10^{11} neurons, organised into different structures and densely interconnected. Different sorts of signals flow throughout the brain, all of them being processed by the neurons almost simultaneously. In this section I will sketch the organisation of the nervous systems, not focusing on details, but rather looking from a structural point of view. For a comprehensive description of the neural system, one might refer to (Kandel, Schwartz & T.M.Jessell 1991, Shepherd 1994, Nicholls, Martin & Wallace 1992).

To describe a system as complex as the human brain, one must start from the very low molecular level. Nevertheless, since the models that will be considered in this work approximate the neuronal system at a very coarse level, the description here will start from a neuronal level.

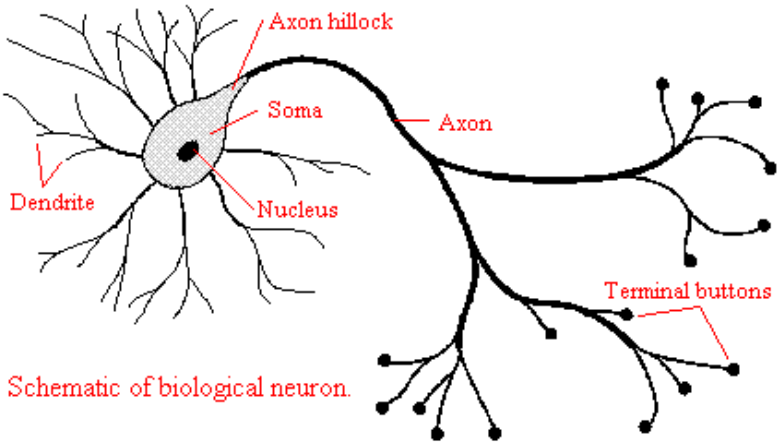

Figure 3.1: Structure of a biological neuron. Functionally there are four main elements: (1) a soma collecting the signal coming through (2) dendrites (input fibres) and distributing the signal further through (3) axons (output fibbers). An axon from one neuron is connected to the dendrites of another neuron through (4) synapses (or terminal buttons), which represent the main memory of the neurons.

Neurons

Biological *neurons* (Fig.3.1) are cells with the following main functional parts: a body (*soma*), which accumulates signals coming from the input fibres (*dendrites*) and which produces at the *axonal hillock* a series of bursts (impulses) when the accumulated signal reaches a critical *threshold*. These impulses are propagated to other neurons through an output fibre called (*axon*). The axon branches and connects to other neurons via connections of variable strength called *synapses*. On average, one neuron receives signal from 1000 other neurons. The synapses are in fact the *Long-Term Memory* (LTM) of the neurons. The synaptic size (conductance) varies in time, in accordance to some biologically laws called learning mechanisms. On the other hand, the current activation of the neuron is its *Short-Term Memory* (STM) and it changes much faster than the synapses.

The signals that are being propagated through the neurons consist of a train of impulses, called also action potentials, or spikes. Those spikes

have a chemic-electrical nature. The basic mechanism that propagates them through the nerve cell is called a sodium-potassium pump (Shepherd 1994). The stronger the signal, the longer the length of this train. The spikes themselves have relatively constant nature – at the place of a spike, the cell is being depolarized from a rest potential of about $-70mV$ to some $+40mV$, for about $1-10ms$. The length of the train of spikes varies from $0-10$ per second which corresponds to almost inactive state of the neuron, to about 1000 spikes per second, which is a very active burst.

There are various types of synapses. Firstly, there are chemical and electric ones, the first type being predominant throughout the nervous system (Nicholls et al. 1992). Electric synapses are much faster, but they are sensitive to small changes in the cells. We find them in places where time is critical, such as the very first layers of the retina. Chemical synapses, on the other hand, are slower since they use chemical (neuro) transmitters to transfer signal and a number of related biochemical processes. Most of them use Glutamate and acetylcholine (ACh) as a neuro-transmitter, with primarily excitatory synaptic action. There are negative connections, too, mostly based on the GABA and Glycine neuro-transmitters. Further, synapses connect not only axons to dendrites, but also axons with axons; axons with somas and even dendrites with dendrites. This means that even at a neuronal level there is a possibility for signal transformation.

Neuronal circuits and systems

The huge power of the nervous system is due to the complex organisation of the neurons, which occurs at various levels. *Local circuits* refer to a group of neurons within certain region whose function is to implement very simple local computations, such as reexcitation; antagonistic interactions, and so on (Shepherd 1994).

The distributed nature of data processing is exhibited at another level of organisation, called *neuronal field* – a set of neurons which perform one step of the processing of one type of signal, with possible inter-connections among them. One or a few neurons in this field process in parallel one feature of the distributed signal (data).

Further, neuronal fields located at adjacent layers in the brain are organised into a *column* – a neuronal structure that performs a complex transformation of the distributed over many neurons input signals. A typical example of this structure are the columns that process visual signals in the brain in the so-called V1 - V4 visual fields. The raw visual signal there is gradually transformed, by extracting features of increasing complexity.

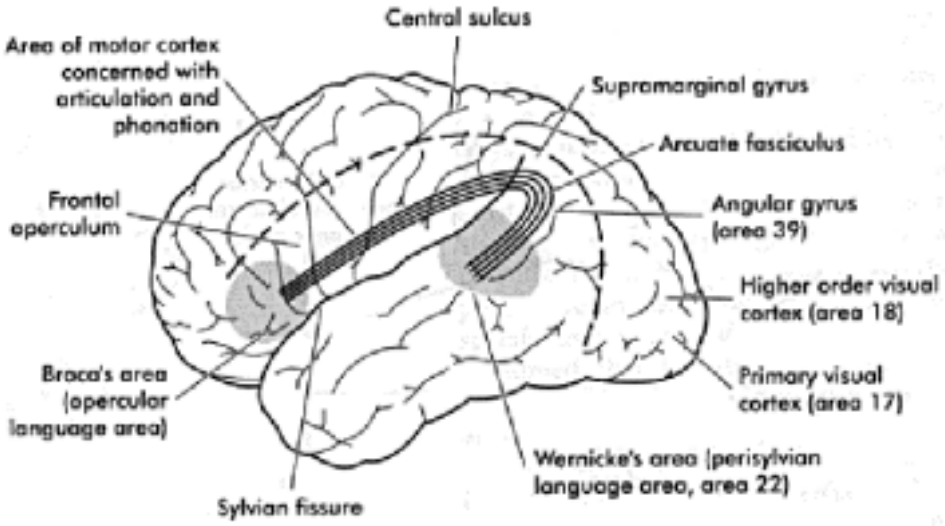

Figure 3.2: A schematic representation of the brain region involved in language processing. Of particular interest are the Broca's and the Wernike's areas.

At still higher level of organisation are the *pathways* and *distributed systems*. The former usually refer to the complete processing of a signal from a certain modality. For example, the visual pathway starts from the retina where the visual signal is detected; goes through the thalamus into the visual cortex, and then through the different visual regions V1, V2, and so on. At every step complex processing is done, with possible associations to other modalities. It is found that in such pathways the signal proceeds not only in a forward direction, but also that there is an extensive feedback which is responsible for processes such as prediction and signal completion.

Distributed systems, on the other hand, refer to a number of connected regions which together mediate certain aspect of the behaviour of the organism. For example, there are auditory, visual, motor systems, and so on. All functional systems build up the complete nervous system and most of them are located into the brain. The systems performing the particularly interesting high-level functions are located in the so-called *cerebrum*, or *neocortex*. A number of brain-imaging studies as well as earlier experiments lead to the general acceptance that language processing is done primarily in

a portion of the posterior aspect of the third frontal convolution (Broca's area) and the region including the posterior aspect of the superior temporal gyrus (Wernike's area) of the left cerebral hemisphere. A lateral view of the left hemisphere, with the language-relevant structures represented, is given in Fig. 3.2.

Learning and Memory

It is the capacity of the neurons to adaptively modify the strength of the synapses and to keep these changes relatively stable that provides the living organisms with the capacity of complex behaviour. The synaptic modification is called *learning*, and it is a dedicated synaptic process that is expressed externally as a change in behaviour and which is caused by experiencing the external environment (world) (Shepherd 1994, Martinez & Kesner 1998). The modification is adaptive because it has some meaning for the behaviour and survival of the organism. We might call the capacity to keep the changes relatively stable as *memory*. Memory also means the capacity to store and recall previous experiences, which is a more complex process. Memory and learning are interconnected. Memory is necessary for learning, it is the product of the learning process.

The main principle of learning at a neuronal level was postulated by Hebb (1949): "When an axon of a cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased". This is to say that any two connected cells that are repeatedly active at the same time will tend to become associated, so that the activity of only one of the cells facilitates the activity in the other. Nevertheless, different types of neurons in the brain have different learning mechanisms. They differ in (a) signals participating in learning, (b) neurobiological levels at which learning takes place, and (c) processes which realise the actual learning. With regard to this, the strength of a connection might have a short-lived increase or it might involve some long-lasting structural change in synapse, as in long-term memory.

The learning at a neuronal level is expressed also at more global levels. For example, what is referred to as *habituation* and *sensitisation* – which represent weakening and strengthing of a connection – is expressed both at a neuronal (synaptic) level and at a behavioural level, where it represents a decrease / enhancement of the animal' response after repeated presentation of a given stimulus (Shepherd 1994). At a high level also other more complex types of learning are distinguished. One of them is *associative learning*,

in which an animal makes a connection between a neutral stimulus and a second rewarding/punishing stimulus. A more specific type of such associative learning is *classical conditioning*, where a conditional stimulus is paired with a unconditional stimulus. In classical conditioning, the animal is passive recipient. By contrast, an animal might be asked to learn a task or solve a problem, for which it is rewarded or punished. This is called *operant conditioning*; since the animal usually makes mistakes before learning the task, it is also called *trial-and-error learning*.

As far as memory at a global level is concerned, there is a growing conviction that the hyppocampus plays a critical role in learning and memory. The central assumption is that stimuli enter the neocortex via the sensory system and subsequently activate the hyppocampus, which plays the role of a (global) Short Term Memory, or episodic memory. The hyppocampus in turn repeatedly provides feedback to the neocortex and gradually initiates activation patterns there, which process is termed memory consolidation (Gluck & Myers 1998). This way, the evolution has solved one very important problem in learning – the capacity to learn new patterns without interfering older memories, what Carpenter & Grossberg (1992) call also the *stability-plasticity* dilemma.

3.4 Artificial Neurons and Neural Networks

The structure of the biological neurons, with a lot of simplifications, is directly used for the development of artificial neurons, as explained in the following subsection. However, the interest here is focused on the structures that can be built with those neurons, since networks of neurons are the models that give the computational power and reliability of connectionism.

3.4.1 Artificial Neurons

The artificial models of the neurons that are exploited here and in most of the literature make a number of simplifications. First, the train of neuronal spikes is replaced by a continuous activation. Since this loses some of the temporal characteristics of the signal, more recent models of artificial neurons – spiking neurons – represent the activation with spikes (Maass 1997). Another simplifying assumption is that the artificial synapses are able both to excite and inhibit the connected neurons, which does not occur that often in the brain. Yet, as it was recently found, the acetylcholine neurotransmitter can indeed produce both excitatory and inhibitory responses in the hyppocampus.

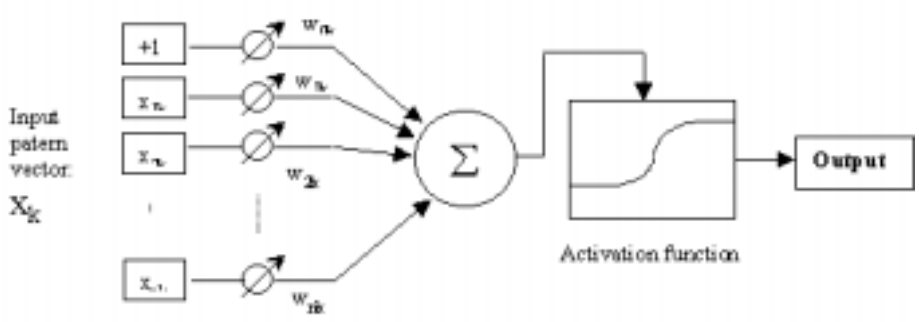

Figure 3.3: A schematic diagram of a McCulloch & Pitts-like artificial neuron. The neuron is activated with a distributed input signal X that enters the neuron through weighted input lines. Then, the accumulated signal is passed through an activation function f() which squashes the signal into the range $0 \ldots 1$. The output of this function is biased by a threshold – a special parameter usually represented with the weight of one of the input lines (the first one) and constant input one.

The first artificial models of the neurons and the brain were proposed by McCulloch and Pitts [1943] (Figure 3.3) . Their artificial neuron is still the basis of today's artificial neurons. It has a vector of input lines $(x_1 x_2 \ldots x_N)$ with variable conductance $(w_1 w_2 \ldots w_N)$ (which stand for the variable synapses) that transmit the incoming signal into a summator S (formula 3.1). The accumulated signal is biased by a rest-potential value or a threshold θ. More complicated artificial neurons might also have internal memory, leaking parameter, etc.

Biological neurons produce an output impulse if the accumulated activation exceeds certain value, which is followed by reducing the current activation. If after a short period (refraction) there is still enough signal, the neuron fires again, and so on, thus producing a train of impulses. As noted earlier, the length of this train of impulses may be interpreted as the strength of the output signal. To model this, the accumulated signal S in the artificial neurons is propagated through an output function which produces the output Y of the neuron (3.2).

The earliest versions of the artificial neurons used the so-called hard-limited output function, which outputs one if the signal S is greater than

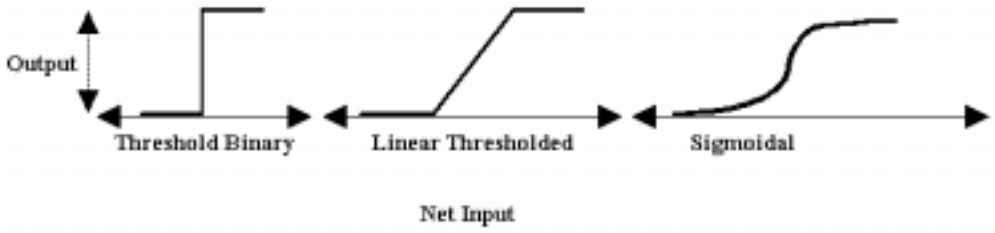

Figure 3.4: Neuron activation functions: a hard-limited function (left), a linear threshold (middle), and a sigmoidal function (right).

zero, otherwise produces zero (see Fig. 3.4, to the left). A more useful activation function is the sigmoid function, which nonlinearly squashes the input signal S into the range $0 \ldots 1$.

$$S = \sum_{i=1}^{N} w_i x_i + \theta \tag{3.1}$$

$$Y = f(S) \tag{3.2}$$

3.4.2 Neural Structures

The artificial models of the neuronal system – the Artificial Neural Networks (NNs), (see Fig.3.5) – consist of artificial neurons organised into *layers* (similarly to the neuronal fields). The neurons in one layer perform the same type of computation, which realise the idea of distributed representations and computations.

A neural network consists of a few layers, the neurons from two adjacent layers being connected in such a way that the neurons from the first layer send a signal to all neurons from the second layer. A convenient method to represent the set of connections from a layer L^A to a layer L^B is to use a weight matrix W^{L^B,L^A}, in which an element $W_{i,j}^{L^B,L^A}$ from a row $W_i^{L^B,L^A}$ represents the synapse connecting neuron L_j^A to neuron L_i^B. A neural network with a few successive layers and this sort of connectivity is called a *feed-forward neural network*. This type of NNs normally process static signals, which do not change in time.

In addition, the neurons within one layer can also be interconnected, which represents a recurrent layer. If the neurons from a layer later in the pathway of the signal provide activation to the neurons from earlier layers, this structure is called a *recurrent neural network* (RNN). RNNs are also those NNs which have recurrent layers. The recurrence induces internal state memory into the NNs, represented as the activation of the neurons providing past signal. This type of networks are used when dynamic data is processed, such as in robot control, speech recognition, general signal processing, etc.

Similarly to sensory receptors in organisms, all NNs have an input layer where external signals are sensed and transmitted to intermediate (hidden layers), which perform the actual NN task. In organisms some of the neurons stimulate effector cells – muscles, glands, etc. Similarly, there is an output layer in the artificial neural networks, whose neurons are meant directly to control physical devices.

In connectionism, the structure of the brain is further reflected in more complex artificial neural network structures consisting of various simple NNs, which perform different functions on the data. Those NN modules are connected in such a way that they perform complex tasks, similarly to large software packages.

3.4.3 Neural Networks Learning

One of the most important property of the Neural Networks is their capacity to learn from their environment, that is, to adapt their long-term memory according to some learning strategy. As already noted, there are many notions associated with the term "learning", but at any rate most of them define it in terms of adaptation aiming at improving the behaviour in the environment (Haykin 1994).

More precisely, NN learning is a process of systematic adjustment of the NN's free parameters (synapses or weights) in order to improve the network's performance in some particular learning environment to acceptable levels. We see here that the learning also involves: (1) an environment providing learning examples, and (2) a designated task.

The learning algorithms in the Artificial NNs fall into three main categories, depending on the information which the training environment provides. If the NN is given both an input signal and desired output signal, this is called a *supervised* learning. On the contrary, if the learning mechanism does not use desired output signal, then the learning is *unsupervised*. Another class of learning mechanisms, which is close to the supervised learning is so-called *reinforcement* learning, in which the environment provides only a

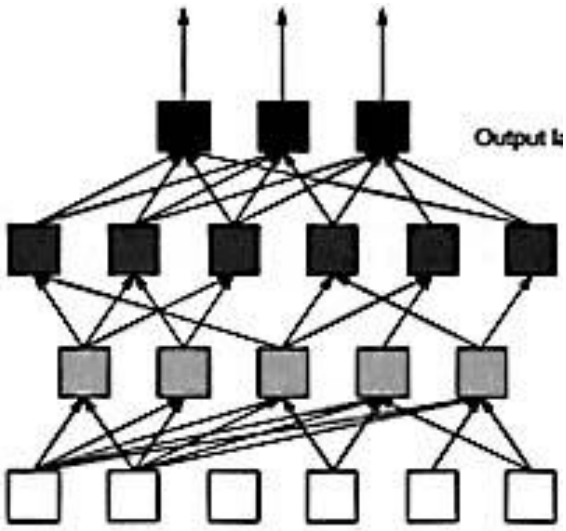

Figure 3.5: An example of an artificial neural network. The neurons (squares) are organised in layers. Each neuron processes one feature of the distributively represented data. The distributed signal enters the network at the input layer (here at the bottom), is consequently processed by all layers and is output at the output layer (here at the top).

positive (rewarding) or negative (punishing) signal to every NN action. The NN can use this signal to encourage or discourage similar behaviour. The Hebbian learning mechanism belongs to the class of unsupervised learning methods. Section 3.7 provides in detail one particular supervised learning algorithm used to train Recurrent Neural Networks.

3.5 Data Representation

An important question concerning the NN implementation is the way the external data is encoded and represented to the networks. The NNs have typically one input layer and one output layer, also called interface layers. In a connectionist systems with an interface layer $N = (n_1, n_2, \ldots, n_{|N|})$, there are two basic methods to represent the items from a set $C = \{c^i\}_{i=1}^{|C|}$.

The more apparent encoding method – orthogonal (localistic) encoding

Symbol	f_1	f_2	f_3	f_4	f_5
a	1	0	0	0	0
b	0	1	0	0	0
c	0	0	1	0	0
d	0	0	0	1	0
i	0	0	0	0	1

Table 3.1: Orthogonal encoding of a set of symbols C_A ={a, b, c, d, i}. There are as many neurons as items in the set, each neuron representing one symbol.

– is similar to the methods of encoding symbols in the symbolic systems with one neuron standing for one item. On the other hand, feature-based representations are truly distributed representations and they exhibit the full power of the PDP. I will explain these in turn.

Localistic representations

To be more specific, localistic representation is defined as follows:

- as many neurons as the number of the symbols (tokens): $|N| = |C|$, that is, for every token c^i there is a representative neuron n_i.
- a neuron n_i is active if and only if the correspondent token c^i is present. The activations of all other neurons are set to zero or other small value.

An exemplary orthogonal encoding of the tokens from a set C_A ={a, b, c, d, i} is given in Table 3.1, where each letter features only one active property $f_1, f_2 \ldots f_5$. This makes difficult for a system to generalise and a fail of one neuron follows incorrect processing of the corresponding item.

Feature-based representations

A representation that is much more appropriate for the connectionist ideas is feature-based encoding. It is based on a set of features $F = \{f_1, f_2, \ldots, f_{|F|}\}$, each of them having some explicit semantics. Interface layers working with these representations have as many neurons as features. A given item c^i is

Symbol	*Vowel*	*Consonant*	*Bi-Labial*	*Dental*	*High*
a	1	0	0	0	0
b	0	1	1	0	0
c	0	1	0	0	0
d	0	1	0	1	0
i	1	0	0	0	1

Table 3.2: Feature-based encoding of a set of letters C_A ={a, b, c, d, i} (rows) with features '*Vowel*', '*Consonant*', '*Bi-Labial*', '*Dental*', and '*High*' (columns). Letters belonging to the same class share common values for features determining this class, while having distinct values of features for other characteristics.

represented as a vector of values for all features. Those values can be binary, or continuous.

Characteristic for this type of representation is that items belonging to the same class of the data share common feature values determining this class, while having distinct values of features for other characteristics. For example, the same set of letters C_A encoded earlier with orthogonal representation is encoded in Table 3.2 with a feature set {'*Vowel*', '*Consonant*', '*Bi-Labial*', '*Dental*', and '*High*'}. In this representation, the vowels 'a' and 'i' share common vowel-specific values for the features '*Vowel*' (active), and '*Consonant*', '*Bi-Labial*', and '*Dental*' (inactive), but differ in the value of feature '*High*' – active for 'i' and inactive for 'a'. Similarly, the consonants agree in features for shared properties and differ in some distinct features.

The feature-based representations exploit all properties of the distributed representations. Firstly, we can encode many more tokens within a limited amount of neurons. The theoretical number of distinct items that can be encoded with k binary features (those that can take two possible states, e.g., 0 and 1) is 2^k, that is, with 10 features we can represent 1024 items. However, in order to get an advantage of some other useful properties such as generalisation, it is reasonable to use not all available combinations of features. Another advantage of this encoding is that once we have chosen the set of features and have started to exploit a model with this set, we can dynamically add other tokens. The network will generalise and work with a reasonable degree of precision even for the new unseen tokens. Further,

just a short extra training would refine the network's work for all tokens. Therefore, wherever it is possible, it is more preferable to use this feature-based encoding. Still, there are some tasks, e.g., in phonotactics learning, where orthogonal representations are preferable to use (see Chapter 4 for details).

Symbol Encoding Scheme

Since the data encoding will play an important role in the rest of this work, I will define here more precisely the encoding of external items with a token encoding scheme. This definition will make use of feature-based representations, but note that the localistic encoding is simply a more specific feature-based representation with dedicated feature for each item.

Definition 4 (Symbol encoding scheme E_F^α)
Let the set of symbols $\{c^i\}$ to be encoded constitute an alphabet $\alpha = \{c^i\}_{i=1}^{|\alpha|}$. Then, a symbol encoding scheme E_F^α of the symbols $c^i \in \alpha$ with a feature set $F = \{F_1, F_2 \dots F_{|F|}\}$ is a look-up table $T(\alpha \times F)$ which associates a vector of feature values with each symbol $c^i \in \alpha$: $C^i = (f_1^i, f_2^i \dots f_{|F|}^i)$.

The item encoding will be used when external data is to be presented to the interface layers of the NNs. However, often we will also need the inverse process, namely to decode the output of the networks. In a pure connectionist system this will not be necessary since the data there is represented only in a distributed way. However, for the purpose of testing connectionist models and in hybrid systems where symbolic methods are used together with connectionist ones, there should be a mechanism that translates the distributed output into symbols.

Such a decoding mechanism will employ the same look-up table $T(\alpha \times F)$ that associates symbols c^i with their distributed representations C^i, but applied inversely, that is, there should be a look-up mechanism that finds that vector C^j among all vectors $\{C^i\}_{i=1}^{|\alpha|}$ from this table, which is closest to the NN output vector Y according to a norm $||.||$ (as yet to be determined), and returns the symbol c^j which corresponds to this vector C^j. Another – connectionist – approach is to use a NN that translates distributed representations into localistic representations.

Internal representations vs. External distributed representations

A more specific subset of distributed representations are the internal representations which the NNs develop in the course of their processing. I

draw attention to this group because later in the thesis, in Chapter 6, I will explicitly make use of those representations in order to recode input data. The internal representations the networks develop do not differ much from the feature-based representations in terms of the truly distributed way of representing data.

They rather differ in terms of 'understandability' to humans. The internal representations are developed by the NNs; they have their meaning specifically for the NN which has developed them and are almost meaningless from a first glance. Yet, some numerical methods such as Principle Component Analysis (Haykin 1994) might extract some information about the representations. In contrast, external distributed representations have more explicit meaning because they usually are designed by the human designer of a connectionist system, or have some natural semantics which is known to us, too.

3.6 NN architectures

After presenting some background information about connectionist systems, in this section I will review shortly a few of the well-established neuronal architectures and will focus on those models which are adequate for language processing and modeling.

The extensive Neural Network research in the last two decades resulted in a number of neural network models (Haykin 1994), which have already found different practical applications. Given a specific type of basic neuron, the functionality of a neural network model depends on its architecture (organisation of the neurons into layers and connectivity among them), learning algorithms and control mechanisms. Those parameters alone provide a big space of possible models, to which finer specifications, such as data organisation, hierarchical architecturing, etc, may be added. Since the purpose of this chapter is only to acquaint the reader with background information relevant to the remaining of the thesis, I will mention quickly just some of the existing NN models and present in detail the so-called Simple Recurrent Networks.

Supervised NNs

The NN models are divided into two basic classes according to their learning method: supervised and unsupervised. Typical models belonging to the first class are the Multilayer Perceptron (MLP) trained with the Error-Backpropagation learning algorithm (Rumelhart, Hinton & Williams 1986)

and the Radial Basis Function neural networks (Haykin 1994). These networks are feed-forward and static.

One of the most famous NN models is the Multilayer Perceptron. It has drawn special attention because it was theoretically (Hornik, Stinchcombe & White 1989) proven capable to approximate any static function provided enough neurons in the hidden layer size, which was also investigated by a number of experimental works (Lawrence, Giles & Fong 1995).

The MLP has one input layer Inp, one output layer Out, and one or more hidden layers $\{Hidd_i\}$ (see Fig.3.5). It is feed-forward: the signal enters the input layer, is propagated to the hidden layer through the weight matrix $W^{Hidd,Inp}$ that connects those two layers, and then goes to the output layer through the correspondent weight matrix $W^{Out,Hidd}$. The hidden layer(s) and the output layer are those whose neurons perform the real computations, the input layer only registers the input signal. The MLP operates in two regimes: (1) use and (2) training. During the first regime, the MLP simply propagates the input signal to the output layer, and the eventual connected devices make use of it. On the other hand, the training regimen involves two main steps: first, a forward step as in the normal regimen of use. After that, the network' output is compared with the desired output activations and an error is computed – the difference between those two. This error signal is then propagated backward through the layers, till the input layer. Therefore, the learning mechanism of the MLP is supervised: an error signal is used to adjust the neurons. One can find the concrete computations describing the BP algorithm in (Haykin 1994, Reed & Marks II 1999), but they are very similar to those in Section 3.7.

Supervised Recurrent NNs

Dynamic problems such as speech processing, robot control, etc., pushed connectionist investigations toward NN models capable of handling dynamic data. The first NN models capable of processing dynamic (sequential) data were still static, encoding limited dynamics by means of a window, shifting over sequential data. For example, the NETtalk model (Sejnowski & Rosenberg 1987), using this approach and a MLP was trained to produce phonetic representations of words. The contextual information there which was required to map the current letter correctly was encoded within a shifting window of size seven – three letters on the left and three letters on the right hand side to the letter to be pronounced. This is an example of the so-called *Finite Impulse Response* filter (FIR), where the system response to a given input is limited to a predefined number of steps.

The first real recurrent models were extensions of the Multilayer Perceptron with recurrent connections. They implement another type of dynamics – *Infinite Impulse Response* – where the input at a certain time influences the system response until the dynamics is externally reset. Several recurrent versions of the MLP were developed. In one of them (Jordan 1986), the network state at any point in time is a function of the input at the current time step, plus the state of the output units at the previous step. In another recurrent model independently proposed by Elman (1988) and Robinson & Fallside (1988) – Simple Recurrent Networks (SRNs) – the network's current state depends on the current input and its own internal state, which is represented by the activation of the hidden units in the previous moment (see Fig. 4.3). This internal state is considered as a context that provides information about the past. The latter model is computationally more powerful, since it has an internal state from which the output is computed, while the context state in the earlier model is directly the NN output. SRNs have been successfully employed for many linguistic and other tasks where the objects have sequential nature (Reilly 1995, Wilson 1996, Cairns, Shillcock, Chater & Levy 1977, Stoianov et al. 1998).

SRNs, the Jordan's recurrent network, and other possible similar architectures belong to the more general class of first-order *Discrete-Time Recurrent Neural Networks* (DTRNNs) (Carrasco et al. 1999, Tsoi & Back 1997), which are very similar to the SRNs, but with an output layer that also possibly receives signal directly from the input layer via another set of connections (matrix). In addition, there might be extra hidden layers between the hidden layer and the output, which slightly increases the computational power of the this class of networks. DTRNNs for sequence processing can be defined as follows:

Definition 5 (Discrete Time Recurrent Neural Network RNN^W) *A discrete time recurrent neural network RNN^W is a layered neural network with:*

(1) an input interface layer Inp, representing one static pattern X at a time and containing $|Inp|$ neurons, each neuron standing for a feature k_i of the input pattern;

(2) an output interface layer Out producing one static output pattern Y at a time and containing $|Out|$ neurons, each standing for a feature f_i of the output pattern;

(3) one or more hidden layers $Hid_1, Hid_2, \ldots$, each of them having $H_1, H_2, \ldots$ neurons;

(4) a network weight space (long-term memory) W representing the con-

nectivity in the RNN. (For the more specific Simple Recurrent Networks, $W = \{W_{(Inp+Hid),Hid}, W_{Hid,Out}\}$);

(5) an inherent internal dynamics represented as a network state Z, such as a global memory (e.g., a context layer Con in SRNs) and/or local internal memory (memory in the neurons).

RNN^W processes sequences of externally presented patterns in the following way:

(I) The internal network state Z is reset to its initial state Z_0 before a new sequence is presented to the network.

(II) Presenting a pattern X to the input layer triggers one network processing step, which consists of: (a) propagating the incoming signal through the network weights W, from the input layer, through the hidden layers, to the output layer, resulting in an output pattern Y, and (b) updating the internal network state Z.

DTRNNs can be trained to learn a set of input/output sequences with different algorithms, such as the Temporal Back-Propagation learning algorithm, or the Back-Propagation Through Time (BPTT) algorithm (Haykin 1994). SRNs were initially trained by Elman with the standard Backpropagation learning (BP) algorithm, in which errors are computed and weights are updated at each time step. While biologically better motivated because of the temporally local weight adjustments, BP is not as effective as the BPTT learning algorithm, in which the error signal is propagated back through time and temporal dependencies are learned better. Since SRN is the connectionist model that will be used throughout the rest of the thesis, section 3.7 provides a detailed technical description of the BPTT algorithm that is used for the network training.

There is also an even more general and powerful class of discrete time neural networks – second order DTRNNs, which feature two-dimensional neuronal input: each input connection is a weighted multiplication between a neuron from the input layer and a neuron from the context layer. Those networks, however, are biologically less well motivated due to this specific connectivity (although there are such type of connections in the brain). Currently, they do not have well studied learning algorithms, either. However, such networks are very useful for compact representation of Finite State Automata in the connectionist paradigm (next section discusses encoding of FSA in first order DTRNNs), e.g., (Carrasco et al. 1999, Omlin & Giles 1996).

Unsupervised NNs and their problem with Recurrence

The other big class of NN models are the unsupervised neural networks. Two very typical examples of such models are the self-organising Kohonen maps (Kohonen 1984) and the Adaptive Resonance Theory (ART) (Carpenter & Grossberg 1992). Those two models offer two very different solutions for self-organising data into categories (clusterisation), but as far recurrence is concerned, they lack inherited capacity of processing dynamic data. The problem is that they organise the data into a sort of localistic way – individual neurons become tuned to specific patterns. In order to provide a general capacity for recurrence, a model should be able to encode any possible sequence, which information to be reused as a contextual information for the following decisions.

Supervised NNs find proper ways to encode such context in limited contextual layers due to the pressure of the error-driven algorithms. Unsupervised networks, however, do not have such a driving force and what they can do is to drive individual neurons to respond to specific input patterns. If those neurons are used to represent the context, then it will be represented in a localistic way. Then, in order to be able to represent a sequence of $|\alpha|$ distinct elements with a maximal length of $|L|$, then $|\alpha|^{|L|}$ neurons would be required. It is easy to estimate that with such an exponential dependency, even large neural networks would quickly run out of neurons. In contrast, distributed context representation, such as in SRNs, allows exponential number of patterns to be encoded into a limited in size context.

Nonetheless, attempts to develop dynamic versions of unsupervised NN models have been made. For example, in order to learn bi-grams for the purpose of phonotactic learning, Cotteleer & Stoianov (1999) extended the ART model (Carpenter & Grossberg 1992) with a context layer. Yet, bi-grams are not enough to solve the phonotactics problem (see Chapter 4 for details) as well as many other sequential tasks. But as just explained, since ART develops localistically represented categories, it is difficult to provide longer-term contextual memory.

The static self-organising Kohonen Map neural network was also extended with recurrent connections, which made the network responses dependent on both the current input and the last neural map activations. Models following this idea are the Temporal Kohonen Map (TKM) by Chappel & Taylor (1993) and Self-Organising Feature Map for Sequences (SARDNET) by James & Miikkulainen (1995), among others.

The methods of encoding dynamics discussed so far employ a *global memory* approach – with dedicated neurons representing contextual information.

Another way to deal with dynamic data is to implement local dynamics in non-specialised neurons (*local memory*), instead. The latter type of architectures vary with regard to the place of this dynamics – in the weights, in the activation function, or both (Lawrence et al. 1995, Tsoi & Back 1994).

Finally, I conclude this section by noting a common problem of recurrent networks: the input data they process has to be linear in the temporal dimension. These networks are able to recognise and classify the temporal sequences they have been trained on (SRNs and Jordan Networks) or they have clustered during the self-organisation process (TKM and SARDNET), but they do not extract more complex temporal features or substructures explicitly. In addition, as the length of the sequences becomes greater, the performance worsens, which has been recognised by a number of authors. Bengio, Simard & Frasconi (1994) showed that learning long-distance dependencies is difficult even for very simple tasks (long strings of a few basic symbols). Miikkulainen & Dyer (1991) emphasised that the required network size, the number of training examples and the training time become intractable as the sequence temporal complexity grows. To cope with this problem, Stoianov (2000*b*) has suggested a method for dealing with sequential data with hierarchical structure by using a hierarchical system of a special NN models – Recurrent Autoassociative Networks, which will be presented in detail in Chapter 6.

3.7 Simple Recurrent Networks

This section will present in detail Simple Recurrent Networks (Elman 1988, Robinson & Fallside 1988) – a recurrent connectionist model that will be used in the rest of this thesis. The section will start with general presentation of the model, will continue with a detailed description of its processing mechanisms and the Back-Propagation Through Time learning algorithm that is used to train the model. Finally, a discussion on the computational capacity of the network will be presented.

Simple Recurrent Networks have the structure shown in Figure 4.3. They operate as follows: Input sequences S^I are presented to the *input* layer, one element $S^I(t)$ at a time. The purpose of the input layer is just to feed the *hidden* layer through a weight matrix, which in turn copies its activations after every step to a *context* layer, which provides another input to the hidden layer – information about the past. Since the activation of the hidden layer depends both on its previous state (the context) and on the current input, SRNs have the theoretical capacity to be sensitive to the entire history

of the input sequence. However, practical limitations restrict the time span of the context information to, e.g., 10-15 steps. Finally, the hidden layer neurons output their signal through the weight matrix connecting the hidden layer to the output layer, to the *output* layer neurons. The activation of the latter is interpreted as the product of the network.

The network is trained with a supervised training algorithm, which implies two working regimens – a regimen of training and regimen of network use. In the latter, the network is presented the sequential input data $S^I(t)$, computes the output $N(t)$ using also the contextual information, and its reaction $N(t)$ is used for the task at hand. The training regimen also comprises a second, training step, which compares the network reaction $N(t)$ to the desired one $S^T(t)$, and which uses the difference to adjust the network behaviour in a way that improves future network performance on the same data.

The two most popular supervised learning algorithm used to train SRNs are the simple Back-Propagation algorithm (Rumelhart, Hinton & Williams 1986) and the Back-Propagation Through Time algorithm (Haykin 1994). While the earlier is simpler because it uses information from one previous time step only (the context activation, the current network activations, and error), the latter trains the network faster, because it collects errors from all time steps during which the network processes the current sequence and therefore it adjusts the weights more precisely. However, the BPTT learning algorithm is also cognitively less plausible, since the collection of the time-spanning information requires mechanisms specific for the symbolic methods. However, this compromise allows more extensive research, and without it the problems which will be discussed in the following sections would require much longer learning time.

Therefore, in the experiments reported here the BPTT learning algorithm will be used. In short, it works in the following way: the network reaction to a given input sequence is compared to the desired target sequence at every time step and an error is computed. The network activation and error at each step are kept in a stack. When the whole sequence is processed, the error is propagated back through space (the layers) and time and weight-updating values are computed. The weights are modified when all time steps are processed in this manner. As noted, this procedure results in faster training than the original simple backpropagation learning algorithm used by Elman (1988) when he introduced SRNs.

The following subsection 3.7.1 describes the BPTT algorithm in detail. The reader is advised to read it, but in case he/she does not, this will not cause problems with understanding the rest of the thesis. Nevertheless, it is

important to read subsection 3.7.2, on the computational power of SRNs.

3.7.1 The Back-Propagation Through Time Learning Algorithm

SRNs have two working regimen: (1) an utilisation of a trained network and (2) network training. The first one is simply applying a forward pass, where the current input signal is propagated forward throughout the network and the current context layer activation is used. After each forward step, the hidden layer activation is copied to the context layer, to be used later. The network utilisation is the same as the forward step in the BPTT, described in the following subsection.

The BPTT learning algorithm itself is more complicated. It includes three main steps. Firstly, a forward pass for all tokens from the input sequence, when all network activations are kept in a stack: (3.3, 3.4, 3.5, and 3.6). Secondly, there is a backward pass through time until the beginning of the sequence, where errors are computed at the output layer and back-propagated through the network layers and through time. Those errors and the stored network activations are used to compute weight update values: (3.7, 3.8, 3.9, and 3.10) Finally, the algorithm ends by updating the weights with the accumulated weight-updating values (3.11). The second step requires that at each time moment but the last, a *future error* be used, processed and back-propagated further through time.

In the next subsections follows a detail description of the forward pass and the BPTT algorithm. Before that, all notations are explained.

Notations

In the following description, $|IL|$, $|HL|$, $|CL|$, $|OL|$ stand for the size of the input, hidden, context and output layers, correspondingly. The input signal provided to the i-th hidden layer neuron and the l-th output layer neurons are noted as $net_i^H(t)$ and $net_l^O(t)$, respectively. Next, $in_j(t)$, $cn_k(t)$, $hn_i(t)$ and $on_l(t)$ stand for the activations of the j-th input, k-th context, i-th hidden and l-th output neurons at time t. Finally, $w_{ij}^{H_I}$, $w_{ik}^{H_C}$ and w_{li}^{O} are the weights of the connections between j-th input neuron and i-th hidden neuron, k-th context neuron and i-th hidden neuron, and i-th hidden neuron and l-th output neuron, respectively.

For convenience, the bias for all layers is encoded as an extra input neuron ($j = 0; i = 0$) with constant activation 1. The activation function $f(.)$ is sigmoidal – the logistic function $f(x) = \frac{1}{1+e^{-x}}$ or the hyperbolic

tangent function.

The training data consist of a set of pairs $\{(S^I, S^O)\}$ – input sequences S^I and correspondent target sequences S^O. Each input sequence S^I has the form $S^I = \langle c_0^I c_1^I \dots c_{|S|-1}^I \rangle$, and the correspondent target sequence $S^O = \langle c_0^O c_1^O \dots c_{|S|-1}^O \rangle$. As it will be explained in Chapter 4, if the network is trained on prediction, the same input sequence is also used as a target sequence, starting from the second element and usually finishing with a special *end-of-sequence* pattern #. Next, if the elements of the sequences c_i are symbols from an alphabet α, they are encoded with an input/output symbol encoding scheme E_F^α before presented to the network.

Forward Pass

Processing a new sequence begins with resetting the context layer by setting all context neurons $cn_{k=1\dots|CL|}(t=0)$ to zero. The sequences are presented to the network one element c_t^I at a time. Each input token is encoded with a pre-specified input encoding scheme $in(t) = E^{Inp}(c_t^I)$. For each token, the forward pass is applied.

The forward pass starts with an activation of the hidden layer in accordance with (3.3 and 3.4):

$$net_i^H(t) = \sum_{j=0}^{|IL|} w_{ij}^{H_I} in_j(t) + \sum_{k=1}^{|CL|} w_{ik}^{H_C} cn_k(t) \tag{3.3}$$

$$hn_i(t) = f(net_i^H(t)) \tag{3.4}$$

which is followed by direct copying of the activation values of the hidden neurons to the context neurons. Next, the signal is propagated further to the output layer, by activating the output layer neurons: (3.5 and 3.6).

$$net_l^O(t) = \sum_{i=0}^{|HL|} w_{li}^O hn_i(t) \tag{3.5}$$

$$on_l(t) = f(net_l^O(t)) \tag{3.6}$$

Backward Through Time Pass

The second step of the BPTT learning algorithm for a given pair of training sequences (S^I, S^O), is propagating the error signal back through the network

and time. We suppose that the forward steps for each token c_t^I in the input sequence are already done, keeping the activations and the target patterns in a stack. At this stage we also need to encode the targeting output tokens, according to the output token encoding scheme: $d_l(t) = E^{Out}(c_t^O)$, for $l = 1 \ldots |OL|$.

Next, error and weight updating values are computed in an earlier time cycle, that is, starting from the last token. Firstly, output neuron errors and deltas are computed with (3.7), in which the second term computes the neuron error, and the first term computes the derivative of the activation function with respect to its input $net_l^O(t)$. Neuron delta's $\delta_l^O(t)$ represent the output error transferred back through the activation function, that is, through the body of the neurons. Further, updates $\Delta w_{li}^O(\tau)$ of the weights connecting the hidden layer to output layer are computed with (3.8). The weight updating rule symbolises the Hebbian learning law that synapses change according to the strength of the signal that enters the synapse (here $hn_i(t)$) and the strength of the signal at the post-synaptic side (here error signal $\delta_l^O(t)$). We use τ to denote a global time index and $\Delta w(\tau)$ to stand for the accumulated $\Delta w(t)$ for all items from the current sequence.

$$\delta_l^O(t) = f'(net_l^O(t))(d_l(t) - on_l(t)) \tag{3.7}$$

$$\Delta w_{li}^O(\tau) = \eta \sum_{t=1}^{|S|} \delta_l^O(t) hn_i(t) \tag{3.8}$$

Provided that the activation function $f(x)$ is the logistic function, its derivative $f'(x)$ with respect to the output $y = f(x)$ is: $f'(x) = y(1-y)$.

Next, deltas and updating values of the weights connecting the hidden layer to the input and the context layers are computed in accordance with (3.9 and 3.10):

$$\delta_i^H(t) = f'(net_i^H(t))[\sum_{l=1}^{|OL|} w_{li}^O \delta_l^O(t) + \sum_{k=1}^{|CL|} w_{ik}^{H_C} \delta_k^H(t+1)] \tag{3.9}$$

$$\Delta w_{ij}^H(\tau) = \sum_{t=1}^{|S|} \delta_i^H(t) n_j(t-1) \tag{3.10}$$

where $i = 1 \ldots |HL|$, $j = 0 \ldots (|IL| + |CL|)$ (0-th neuron represents the bias) and $n(t)$ is a joined vector containing both $in(t)$ and $cn(t)$. In (3.9), in addition to the error that comes from the output neurons at the current

time t (the first sum), there is also error coming from the *future* step $(t+1)$, represented as the second sum. More precisely, the latter represents the context layer delta-term $\delta_k^C(t)$, computed by back-propagating the future hidden-layer deltas $\delta_i^H(t+1)$ through the weights connecting the context neurons to the hidden neurons. Finally, all weights are updated according to (3.11) with the accumulated weight-updating values, computed with (3.8 and 3.10).

$$w(\tau) = w(\tau - 1) + \Delta w(\tau) \tag{3.11}$$

Back-propagation learning algorithms are known to run the risk of getting stuck in local minima on the error surface. There are number of techniques designed to overcome this problem. The most useful technique is to apply a momentum term α to (3.11), as it is done in (3.12). The momentum term keeps the movement over the weight error space for some time, even if the network has fallen into a local minimum. Usually, $\alpha = 0.7$.

$$\Delta w'(\tau) = \alpha w(\tau - 1) + (1 - \alpha) w(\tau) \tag{3.12}$$

Another technique that has similar effect is initially to apply higher learning coefficient η and next, to decrease it gradually. This implements a quick rough initial search for the region where the global minimum is located. Later, the exact location of the error minimum is searched with smaller steps. Usually, the initial $\eta = 0.2$ and the decrease might be exponential with a very small step (e.g. 0.9995).

For further reading about SRN, BP and BPTT and other recurrent learning algorithms, one can refer to (Haykin 1994, Reed & Marks II 1999), among others.

3.7.2 Computational Power of Simple Recurrent Networks

In this section I will present shortly a few studies on the computational capacities of SRNs, which basically show that this connectionist model is approximately equivalent to *deterministic Finite State Automata*, but it also features all useful properties of connectionism, such as noise resistance and generalisation. In addition, experimental works show that SRNs can also learn limited-depth context free languages.

Experimental works

SRNs were first experimentally demonstrated to be able to learn to closely mimic deterministic FSA, both in terms of behaviour and state representations. In all training procedures, the networks were trained by examples generated by the FSA. In particular, Cleeremans, Servan-Schreiber & McClelland (1989) and Ghosh & Karamcheti (1992) succeeded in training SRNs to learn the so-called Reber grammar – representing a regular language with four intermediate states, three close loops, and up to two allowed successors at each state – but they had difficulties in learning the Embedded Reber grammar – a more complex version of the former one. Later, Manolios & Fanelli (1994) demonstrated the capacity of an extended SRN – with extra recurrent connections at the output layer – to learn several Tomita regular grammars, e.g., $(10)^*$. In those works, the authors also used cluster analysis to determine the internal representations of their networks and they showed that the neural states tend to cluster into clusters that roughly correspond to the states of the deterministic FSA that has been approximated. However, the equivalence showed was very limited since the performance of SRNs significantly degraded as the string length increases – a problem extensively studied latter by Bengio et al. (1994).

Learning languages generated by context free grammars was shown by Elman (1988) and (1991) in his attempt to explain how the complex natural languages are accommodated by fixed-resource connectionist systems and correspondingly, what is the nature of the language representations developed during training in the context state of the SRNs. The language he explored was formed from a lexicon of 23 items and was generated by a small context-free grammar with several rules forming sentences with relative clauses, number agreement, and various direct object requirements. The training set consisted of 10,000 sentences, each of them containing up to 16 words. After the training, Elman analysed the hidden layer activations and found that without any prior linguistic information, the network learned to predict the category of the words that may follow the left context presented to the input so far. That is, the network discovered the grammar underlying the surface form experienced during training.

Looking for optimal language learning architecture, Lawrence, Fong & Giles (1996) used variety of learning models, including SRNs, and trained them to classify 552 positive and negative examples of English sentences as grammatical or ungrammatical. In contrast to the Elman's experiments, the category of the words here was explicitly represented. An important outcome of this experiment was that SRNs were found to perform best:

they learned to classify correctly the whole training set and scored 64% / 74% on the testing set.

Theoretical works on representational capacity

Latter studies, such as (Kremer 1995, Alquezar & Sanfeliu 1995), theoretically showed that SRNs can encode any deterministic FSA. Those studies view SRNs, and in more generally Discrete Time Neural Networks (DTNNs) as neural state machines: all possible states of all hidden units (and the context ones) are discretized into discrete regions (states) and at each time a new input symbol (pattern) is applied, a new state of the hidden units is computed by using the previous state and the current input symbol. In particular, Kremer (1995) showed how to represent FSA in SRNs with a threshold activation function, while Alquezar & Sanfeliu (1995) has shown that this can also be done with a sigmoid activation function, provided that rational numbers were returned. Those proofs are constructive, in that they have shown how to construct a SRN which simulates a given FSA. However, in order to be able to encode any FSA, those works rely on converting a deterministic FSA into a new FSA in which all original states are multiplicated as many times as there are symbols in the input alphabet of the original FSA. In the new automaton, each new state represents (a) the corresponding state of the original automaton and (b) the input transition due to the corresponding past input symbol.

Other works regard SRNs and their computational capacity as a subclass of the more general first-order Discrete-Time Recurrent Neural Networks (DTRNN). The latter models have a more general set of connections than SRNs do, for example connections from the input layer to the output layer. Omlin & Giles (1996) and Carrasco et al. (1999) applied the approach outlined above in order to prove that fist-order DTRNNs, and in particular SRNs with sigmoidal units and continuous activation can encode any deterministic FSA, and in particular finite-state machines that act as transducers $T(L_{Inp}, L_{Out})$ (Mealy or Moore machines) that translate stings of an input language L_{Inp} and produce strings of equal length from an output alphabet L_{Out}.

Theoretical works on learnability

Proofs about the theoretical representational capacity still do not guarantee that a learning process would end up with an optimal neural network. For example, error-driven learning algorithms might not be able to converge

due to the recurrence or the error surface might be too-complex and cause the network to fall into local minima. In that respect, theoretical works on learnability of RNNs are very important. In addition, when DTRNNs are trained to mimic FSA, there is a problem related to the stability of the contextual memory viewed as a state of an FSA. The recurrent connections tend to drive the activations of the context neurons away from states representing FSA states, and correspondingly the trained network would not behave anymore like the trained FSA.

Kuan, Hornik & White (1994) address the learnability problem, devoting a study to the convergence of the backpropagation learning algorithm training recurrent neural networks with hidden-layer recurrence (SRNs) or output-layer recurrence (Jourdan RNNs). The learning task there is approximating $E(Y_t|\langle X_0 \dots X_t\rangle)$ – the conditional expectation of Y_t given the history of input data $\langle X_0 \dots X_t\rangle$ – by a parametric function (recurrent neural network) $RNN(\langle X_0 \dots X_t\rangle, W)$, as the parameters W range over a parameter space. The training data are sequences $\{Z_t\}$ of random vectors $Z_t = (X_t, Y_t)$, where Y_t is scalar and X_t is a vector. The study proves a theorem about the convergence of the learning process to a weight set W^* that minimises the network approximation error given a few important assumptions, which approximately look like this: A1. The training data – sequences $\{Z_t\}$ of random vectors $Z_t = (X_t, Y_t)$ – are generated by a stochastic process which is "near epoch dependent" on a bounded underlying mixture process and have limited memory (intuition says that Finite State Automata belong to this category of processes, which was also confirmed by one of the authors of this paper). A2. Continuous differentiable error function of second order on the parameter space W, the input sequence (weak assumption), and on the state variables. A3. The recurrent process is a contraction mapping – to avoid chaotic behaviour. A4. The training algorithm is such that (a) it includes recursion on the parameters (weights) in the recurrent neurons (such as the BPTT learning algorithm), (b) the weight set is kept bounded, and (c) the learning coefficient η_t is such that $\sum_{t=0}^{\infty} \eta_t^2 < \infty$ and $\sum_{t=0}^{\infty} \eta_t = \infty$. A5. There is a limit to which the learning process would converge (The above studies on the representational power of SRNs guarantee that there is such a limit). If those assumptions hold, then they apply a theorem proven earlier which roughly states that the gradient learning process converges to a weight set W^* that minimises the network approximation error. For the particular case of learning Elman's SRNs on data generated by FSA, assumption A2 holds and A3 is replaced by Assumption B3 which requires that the recurrent weight vectors are bounded. Using the notation presented earlier in this section, assumption B3 looks

like this: $(\sum_{i=1}^{|CL|}\sum_{k=1}^{|CL|}(w_{ik}^{H_C})^2)^{1/2} \leq 4(1-\epsilon)$, for some $\epsilon > 0$. To guarantee this, the weight updating rule of the learning algorithm (3.11) is extended with a restricted mapping that ensures assumption B3.

Arai & Nakano (2000) consider the stability of a SRN trained to simulate an FSA. The problem is that even if the learning process converges to an optimal solution – a DTRNN that learns the training set of sequences perfectly – this does not guarantee that the DTRNN would mimic the FSA inferable by the training sequences, which as I noted above is because the continuously activated neurons might drive away from regions of states (orbits) that represent FSA states. The works cited above about the representational power of SRNs to encode any FSA deal with this by setting the weights with proper values. Arai & Nakano (2000) solve this problem by (a) causing the sigmoidal nodes to operate almost as threshold units, which keeps the activation of the recurrent nodes near to the corners of a hypercube (one's and zero's), thus stabilising the context memory, and (b) introducing a prior to the BPTT learning algorithm by adding an internal representation term to the optimisation function of the learning algorithm.

Thus, the former study deals with the learnability of training sequences, and the latter one provides a method to extend the learning in a way that would produce SRNs that stably mimic FSA, that is, it guarantees generalisation: good performance for all sequences generated by the corresponding FSA. Nevertheless, the work on the learnability with BP algorithm still does not provide a solution to the local minima problem, which is normally solved with heuristic approaches.